Continuous Quality Assurance in Implemented Architecture Models

Jan Hinzmann
Deutsches Zentrum fuer Luft- und Raumfahrt e.V. (DLR)
Lilienthalplatz 7
38108 Braunschweig
Jan-Oliver.Hinzmann@dlr.de

Axel Berres
Deutsches Zentrum fuer Luft- und Raumfahrt e.V. (DLR)
Rutherfordstrasse 2
12489 Berlin
Axel.Berres@dlr.de

Andreas Schreiber
Deutsches Zentrum fuer Luft- und Raumfahrt e.V. (DLR)
Linder Hoehe
51147 Koeln
Andreas.Schreiber@dlr.de

Daniel Luebke
Leibniz Universitat Hannover
Welfengarten 1
30167 Hannover
Daniel.Luebke@inf.uni-hannover.de

ABSTRACT

During the ongoing development of software the implementation drifts away from the initial design due to the changing requirements (known as *moving targets*), lack of communication and unforeseen difficulties. This drift is unacceptable as the design represents the *common picture* and is the basis for further development and maintenance of the software, and should therefore only be changed after communication took place which updates this common picture. Automating continuous comparison of design and implementation (resulting in *model differences*) during the development phase increases the quality of a software project. Checking continuously for model differences and propagating them appropriately will increase the communication between the roles engaged in software projects leading to less misunderstandings and decreasing the amount of failed software projects.

Our approach detects differences between architecture and implementation in a *non-invasive* manner. It initiates communication between architects and engineers so that both the implementation and the architectural artefact keep up-to-date.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Design

Keywords

Software engineering, static analysis, communication, Java, model differences

1. INTRODUCTION

Whenever people work together, communication plays an important role for the outcome of their work. The area of computer science is no exception. But communication also carries the risk of misunderstanding. Very often, people engaged in projects are not aware of the misunderstandings for a while. The longer these misunderstandings are not detected, the more costly they can become.

As software projects require different roles (customer, architect, engineer, ...) and due to the complexity of many domains, the communication between these roles is essential to the project's success or failure. We see the whole software development process as permanent communication about a problem. In consequence, the quality of communication between the partners in a project can be directly related to the quality of the project's outcome.

After gathering the requirements in software projects, the architect works out an initial design. Afterwards, she has to communicate the design to the developers who will implement it. During the communication the architect's picture of the possible solution gets transferred to the developers. But on the way from the architect to the other team members the picture changes slightly due to the problem of communication (different experience, socialization, etc.). Even if we have a layer of indirection like a design language such as UML, there is still a risk of misunderstandings.

Already at this early stage of development we observe differences between what the architect has in mind and what the developers do. Going one step further, the developers implement their point of view which leads to further differences as they change the design during the implementation.

The result is a software which differs from the architect's model, from the developer's model and (this seems to be most important) there is a high chance, that the project outcome differs strongly from the model the customer had in mind. This is especially bad, if the software design was part of the contract, e.g. if software components are subcontracted.

The above implies that good communication between all roles in a projects is the key to project success. But communication can and often must be initiated. Often one doesn't even know that there are problems in a project. Therefore, we introduce the tool `MoDi`, which compares the implementation to the design, finds out the relevant differences, and communicates the results to the responsible people. Consequently, architects and developers can get into contact and discuss the changes. After such a meeting, either the architect improves the design because the changes were needed or the developers reimplement parts of the software according to the architecture.

The rest of this paper is organized as follows: Section 2 presents an overview of closely related work. In Section 3, we define the relevant terms and describe the basic techniques for comparing software models. Section 4 introduces the current design considerations for a software toolkit for checking model differences. In Section 5, we conclude the paper by summarizing the contributions and possible future work.

2. RELATED WORK

The approach in [5] uses source code annotations and therefore has to be applied during the process. Our approach is non-invasive and can be used even for completed projects.

The tool Sotograph (http://www.software-tomography.com) seems to have a quite similar approach but at the moment we are not aware of the technologie behind it. As sotograph can discover many different issues in software projects our tool `ModI` can be integrated in the daily development cycle.

We agree with [1] that software architecture is a ... *high-level organization as a collection of interacting components, connectors, and constraints on interaction, along with additional properties defining the expected behavior.*

We do not restrict our tool on some specific technology such as ArchJava in [2]. We work on the language itself and therefore our approach is feasible as long the programming language has a defined grammar (e.g. in EBNF). This is needed for generating language specific parsers to import the engineer's model into our meta model. Using a CASE tool the architect should be able to provide a code fragment of his design. This is done by drawing UML diagrams and exporting them to source code or into an XMI representation. Currently, we support finding structural aspects of software but plan to take behavior into account.

3. MODEL DIFFERENCES

In many cases, the architecture of software systems is specified in UML and the implementation on the other side is represented through source code. Due to the nature of the software development process both, the architecture and the imp lementation, are subject to change as the development of the system proceeds. Mastering these changes is a key condition for the success of the underlying project. To assist developers, architects and technical project leads, we propose a tool based approach to discover differences between the intended software design and the implementation continuously. The proposed tool initiates communication by sending reports to associates roles.

The current work covers the structural part of a software system. We have focused on the interfaces defined by an architect as initial design so far.

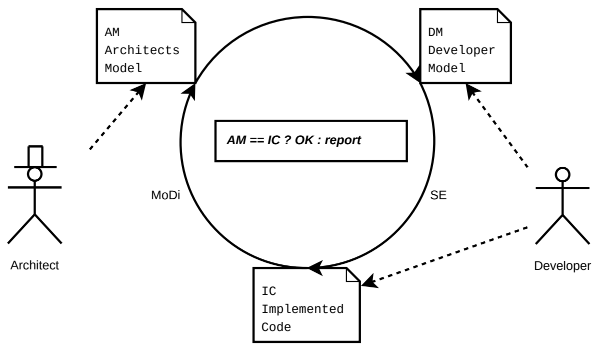

Figure 1: Genesis of model differences

From a high level view of the software development process (see Figure 1) an architect specifies his architect model (AM) using UML. Afterwards, he communicates his design to the engineer resulting in an developer model (DM). Due to the nature of communication, we already observe differences between the AM and the DM after this step.

In the course of the project, the developer implements the DM resulting in the implemented Code (IC). Again we observe differences between the DM and the IC due to the evolutionary process of software development.

It is fundamental to check the resulting IC against the initial AM and discover the differences between them. Initiating communication between the participating roles, leads to a refinement of the AM and to refactoring of the DM/IC.

Applying continuous checking for model differences to the software process improves the communication and in consequence the quality of the whole project.

The tool `MoDi` integrates seamless into many known development processes as it is transparent to the user. Once the architect and the engineers have defined a set of rules they want to apply to their project, `MoDi` checks them continuously and initiates communication gently if a rule has been broken.

3.1 Base of Comparison

As the architect specifies his design in UML using a CASE tool, we get – depending on the serialization of this tool – XMI files or source code. On the other hand we have the code base from the developer model. In order to compare the AM with the DM we have to provide a common base for comparison. It is the responsibility of a transformation component to provide such a comparison base.

3.1.1 XMI

The XML Metadata Interchange (XMI) as OMG standard is used for an interchange format for UML models. Although it is supposed to be understood by the different UML tools, in practice this as rare. Nevertheless `MoDi` can be easily adapted to support different kinds of XMI formats for transforming architectural descriptions into its internal meta representation.

3.1.2 Source Code

More reliable than the XMI representation is the source code and in the end the source contains all of the implemented design. If we have a grammar for the given language dealing with source code becomes very easy. From the EBNF grammar we can create a parser (using tools like lex, bison, JavaCC, SablsCC, GOLDParser or ANTLR) and build an abstract syntax tree (AST). Currently `MoDi` uses the ANTLR parser generator. Having an AST representation of the AM and the DM leads to algorithms for tree comparison.

3.1.3 Meta Representation

Introducing an abstraction layer to the language specific source code let us deal with more than one programming language. Furthermore, this abstraction layer helps us to control exactly the amount of information we need. The `MoDi` system internally generates abstract syntax trees from given source code. Afterwards, it reads the relevant information from the generated ASTs and stores them in the meta representation. Figure 2 shows the situation after an AM and a DM have been parsed and the ASTs have been generated. In the following step, the `MoDi` system extracts the relevant information and sets up the meta representation, which represents the layers of the AST (Model, Item, Member and Token layer).

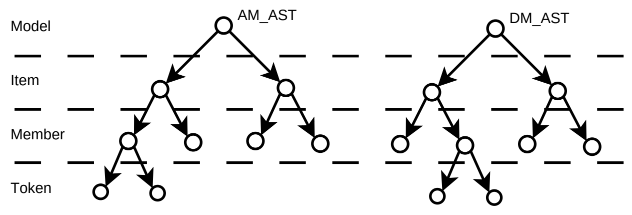

Figure 2: The resulting AST and its layers

Figure 2 also shows model differences. The internal meta representation has to cover all items that are subject to possible changes. Also some meta data on the models can be saved, so that we can reference roles and revisions of the models. The Entity Relationship diagram shown in Figure 3 gives an overview of the current design of the meta representation and what can be.

3.2 Changes

3.2.1 Items of Change in Architectures

Relevant items in modern object oriented languages regarding architectural elements on the source code level are classes and interfaces. Interfaces can be divided into a signature

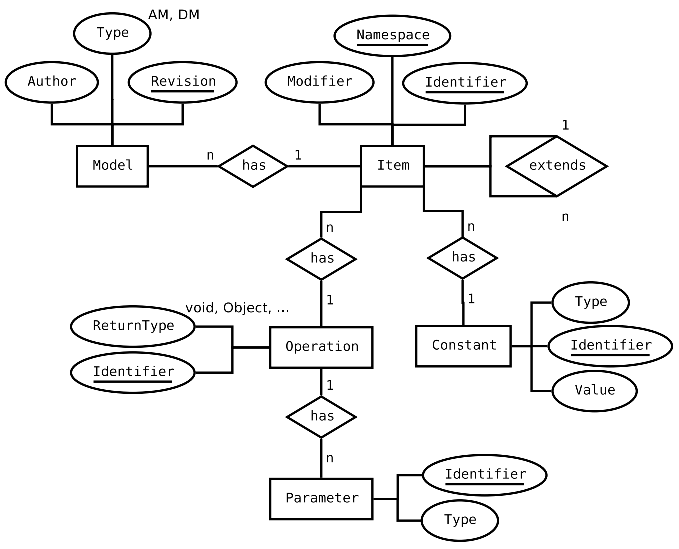

Figure 3: Entity relationship diagram of the meta representation

and a body. For example, the signature of a Java interface is composed of modifiers, the keyword `interface`, an identifying name and a optional list of extended parent interfaces. The body of an interface can contain constants and method signatures. They all have also modifiers, a type and an identifier. Constants then have a value and methods have a return type and an optional parameter list. All these items are subject to change and can lead to model differences.

3.2.2 Rules for Changes

In principle, `MoDi` can identify all differences between the AM and the DM. However, as reporting all differences could result in a great amount of information, the user should be able to define a set of rules that filter this information. These can be something like

- methods (added/removed/renamed),
- parameter (added/...),
- return types,
- names, or
- interfaces.

Although the system could try to be aware of renaming interfaces, we state that this is equal to the removal of an original interface and the addition of a new one. This is because the original interface is not available to any other software component accessing it – therefore, it is deleted from the point of view of that component. Instead, a new interface is visible.

4. TOOL FOR MODEL DIFFERENCES

After having identified the items which can change in architectural elements (such as interfaces), we provide a set of rules which are associated with a specific item of change.

Once triggered by a recognized model difference, the MoDi system will automatically initiate communication between the relevant roles. This communication might lead to a change of the design or not. In any case the common picture of the software is updated for every team member.

4.1 Overview

The principal structure of the tool MoDi is shown in Figure 4.

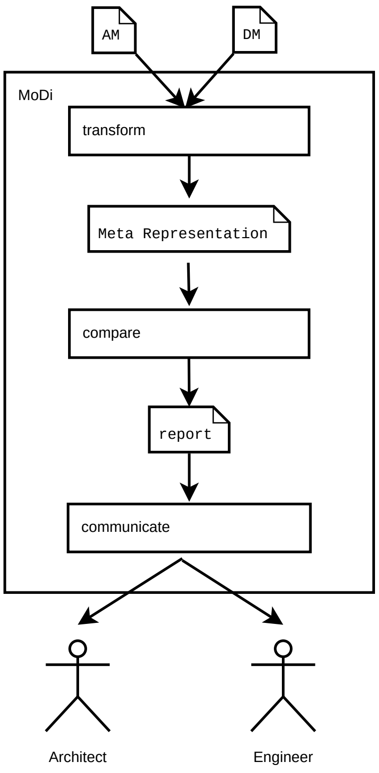

Figure 4: the big picture of MoDi

Starting from a model (AM or DM) the system imports the relevant files and converts them into an meta representation. We found it important to use a meta representation so that we can avoid performance issues that would result from using tree comparison. Using the meta representation the system finds all model differences and can evaluate the configured rules in order to generate a report. If a rule is evaluated to false, the comparison component generates a report item and adds it to the resulting report. This report now holds all report items obtained from the comparison component. Finally, the communication component can apply filter and grouping functions on the report items and send the resulting final report to the associated roles such as architects and developers.

A layered view of the system is shown in Figure 5. It also shows the possibility to implement different transform and parsing components enabling MoDi to work on different languages.

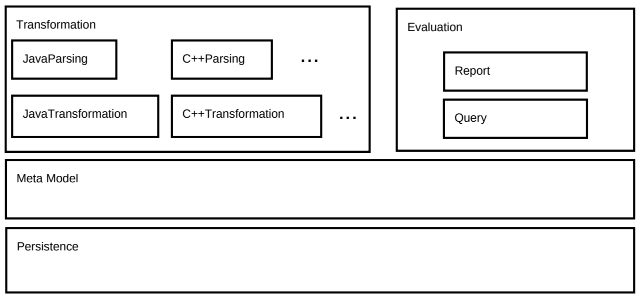

Figure 5: Layers of MoDi

4.2 A Closer Look at the Components

In Figure 4 we have seen the big picture. We will now take a closer look at each of these components. Starting with the transform component (shown in Figure 6) we see that this component has to know where to find a model.

4.2.1 transform

The path to the code bases of the AM and the DM have to be configured in the system. The FileFinder then finds relevant project files using a FileFinderFilter (omitted in Figure 6). Having found all relevant files from the model, the FileFinder passes a list of these files to the Parser component, which parses each file and generates an abstract syntax tree (AST). The resulting trees are then put together in a single model AST (MAST).

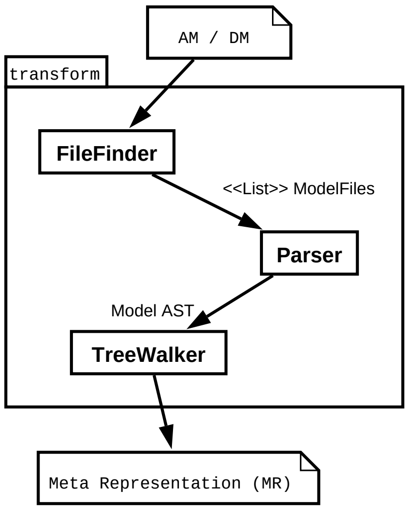

Figure 6: the transform component

The MAST itself is passed to the TreeWalker component which is responsible for the creation of the meta representation. Primarily, the TreeWalker traverses the tree and separates relevant information from irrelevant information regarding model differences. This is reducing the problem space.

4.2.2 compare

Having imported a model using the import component we obtained the meta representation which contains all relevant information for identifying model differences.

The comparison can now be achieved by the compare component (Figure 7). Therefore it queries the meta representation of the architects model and the one provided by the developer. This is done by the `Comparator` object which bases its queries on a set of `Rule` objects configured by the architect. A rule for instance can allow or deny the renaming of interfaces, the addition of methods to or the removal of methods from an interface and also changing return types or type and number of parameters defined in a method signature.

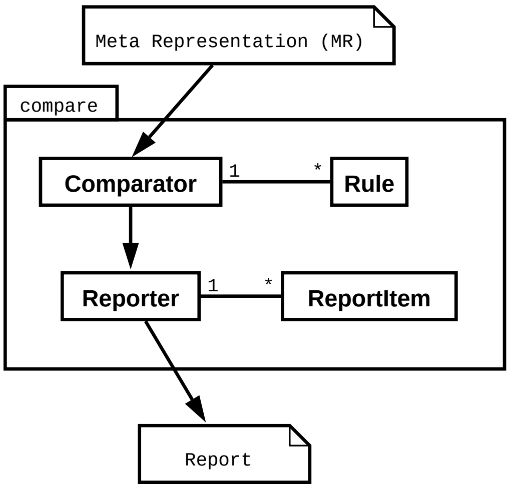

Figure 7: the compare component

However, once the rules have been evaluated, each rule results in a `ReportItem` which is a human readable string representation of its outcome. These report items are collected by the `Reporter` object. Having collected all `ReportItems` the `Reporter` object makes the final report available to the subsequent communication component.

4.2.3 *Communication*

After the comparison of the two models took place, the results must be communicated to the associated roles. This is done by the communication component (Figure 8). One can personalize the report according to the different roles as an architect might want to read other information than a developer. Such role dependent reports could be achieved by filtering and grouping report items according to the role of an architect or a developer but this seems contraindicated as it might lead to misunderstandings if architect and developer discuss upcoming model differences based on different reports.

5. CONCLUSION AND FUTURE WORK

We have implemented a prototype of `MoDi` which is based on the widely used programming language Java. We can translate Java source code into our meta representation. To enable support for other languages one simply has to follow these steps:

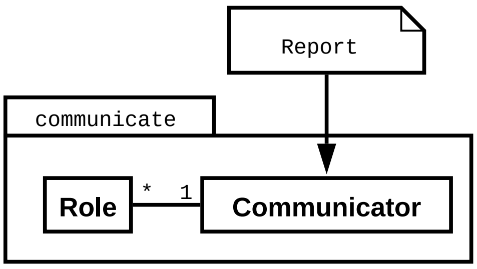

Figure 8: the communicate component

1. write or obtain an ANTLR grammar for the language,
2. generate the language specific parser using ANTLR,
3. implement a TreeWalker in order to retrieve the relevant information from the ASTs
4. define a set of rules.

Using the tool `MoDi` continuously during a software project, one is able to detect model differences just at the moment when they appear (they manifest themselves during a commit to the SCM). Therefore, we are aware of the drift from the initial design and can initiate communication among the involved roles at an early stage of this drift.

Having this prototype of `Modi` we will use it to discover model differences in a software project that has already finished. In order to simulate the progress of that project we retrieve svn revision from the projects repository. Like this we can messure the amount of changes in the architecture and the implementation. We also apply the `MoDi` tool to current projects in order to improve our system.

6. ADDITIONAL AUTHORS

7. REFERENCES

[1] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, and Tony Tseng. Improving system dependability by enforcing architectural intent. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE*, pages 187–197, 2002.

[3] Walter R. Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph - a pragmatic approach to source code architecture conformance checking. In *EWSA*, pages 1–9, 2004.

[4] Won Kim. On designing software architectures. In *Journal of Object Technology , vol. 5, no. 7, September-October 2006, pp.27-32*, 2006.

[5] Holger Krahn and Bernhard Rumpe. Enabling architectural refactorings through source code annotations. In *Modellierung*, pages 203–212, 2006.

[6] Roshanak Roshandel, Bradley R. Schmerl, Nenad Medvidovic, David Garlan, and Dehua Zhang. Understanding tradeoffs among different architectural modeling approaches. In *WICSA*, pages 47–56, 2004.

[7] Friedrich Steimann and Philip Mayer. Patterns of interface-based programming. *Journal of Object Technology*, 4(5):75–94, 2005.